Estudiante: Federico Terán Pascuas

Código: 8977473

Profesores: Gonzalo Noreña y Carlos Ramírez

Curso: Estructuras de datos

Carrera: Ingeniería de sistemas y computación

Semestre: Segundo

Reporte de complejidades

Operación: addLogic

Complejidad: O(n)

Análisis: La complejidad computacional de la operación O(n) donde n es en el mejor caso el tamaño del BigInteger más pequeño que se va a sumar y en el peor caso n es el tamaño del BigInteger más grande. En el algoritmo se recorre con un ciclo iterando las veces del BigInteger más pequeño y sumando al contenedor deque<int> deq lo que se encuentra posición a posición de los dos BigIntegers a sumar y adicionalmente un acarreo de la suma, en el mejor caso si al objeto al que le aplicamos la operación tiene un size más grande que el BigInteger que se pasa por parámetro entonces el algoritmo daría fin una vez termine de sumar el acarreo restante al contenedor deq. En el peor caso, si el objeto al que le aplicamos la operación tiene un size menor que el objeto que se pasa por parametro entonces después de sumarle al contenedor deq posición a posición de los BigIntegers hasta que se recorra por completo el menor, se procede a hacer push_back() al contenedor deq las cifras restantes que se encuentran en el objeto dado en el parametro. En el primer caso descrito (el mejor caso) n corresponde a el size del menor, mientras que en el segundo caso (el peor caso) n es el size del BigInteger más grande.

Operación: substractLogic

Complejidad: O(n)

Análisis: En la operación substractLogic se cumple que se invoca únicamente cuando el objeto al cual se le va a aplicar es mayor que el BigInteger que se le va a pasar como parámetro. La complejidad de la operación es O(n) donde n en el mejor caso es el tamaño del BigInteger que se pasa por parámetro y en el peor caso n es el tamaño del BigInteger al que se le aplica la operación. El mejor caso esta dado cuando cada cifra del sustraendo es menor o igual que la del minuendo, de esa manera la resta se podría realizar restando posición a posición del BigInteger en un ciclo que itera las veces del objeto pasado por parámetro. En el peor caso n puede llegar a ser el tamaño del BigInteger más grande (que siempre va a ser el BigInteger al que se le aplica la operación) este caso ocurre cuando las cifras del minuendo son menores que las del sustraendo, por lo que se cae en un ciclo que recorre el BigInteger más grande hasta encontrar una cifra que le pueda prestar al minuendo que necesita.

Operación: addCeros

Complejidad: O(n)

Análisis: La complejidad computacional de la operación se realiza en tiempo O(n) donde n es el número entero dado como parámetro de la operación. El algoritmo entra en un ciclo que itera n veces,

en cada iteración modifica el objeto al que se le aplica, adicionando un cero al final del BigInteger en cada iteración.

Operación: get

Complejidad: O(1)

Análisis: La complejidad computacional de la función se realiza en tiempo O(1). La operación se realiza en tiempo constante ya que esta solo devuelve el entero que se encuentra almacenado en el contenedor deque<int> deq en la posición que se le está pasando por parámetro.

Operación: size

Complejidad: O(1)

Análisis: La complejidad computacional de la función se realiza en tiempo O(1). El tiempo de ejecución es contante ya que esta operación solo devuelve el tamaño del BigInteger, esto se realiza aplicándole la función size() al contenedor deque<int> deq que es donde se almacena el BigInteger.

Operación: getSign

Complejidad: O(1)

Análisis: La complejidad computacional de la función se realiza en tiempo O(1). La función se da en tiempo constante porque solo retorna el signo del BigInteger al que se aplica la función, el cual se encuentra almacenado en un booleano de la clase.

Constructor: BigInteger()

Complejidad: O(1)

Análisis: La complejidad de este constructor se da en un tiempo de ejecución O(1) ya que solo crea un objeto BigInteger, sin embargo ni inicializa ninguno de sus valores.

Constructor: BigInteger(const string &cad)

Complejidad: O(n)

Análisis: La complejidad computacional de aplicar este constructor se siempre en tiempo O(n) donde n es el tamaño del string cad que se pasa por referencia como parámetro, el algoritmo recorre cad y transforma cada uno de los caracteres numéricos en enteros y después los agrega al contenedor deque<int> deq.

Constructor: BigInteger(const BigInteger &bigInt)

Complejidad: O(n)

Análisis: La complejidad computacional de aplicar este constructor se realiza en tiempo O(n) donde n es el tamaño del BigIntger pasado por referencia en los parámetros del constructor. Al aplicar este constructor el deque<int> deq se inicializa haciendo una copia del deque<int> contenido en BigInt, hacer una copia del deque implica recorrer este mismo copiando todos sus valores lo cual se da en un tiempo lineal.

Constructor: BigInteger(const deque<int> &d)

Complejidad: O(n)

Análisis: La complejidad computacional de la operación se realiza en tiempo O(n) donde n es el tamaño del deque<int> pasado como parámetro, aplicar este constructor implica inicializar al contenedor deque<int> deq igual al deque d pasado como parámetro, para que esto se dé se realiza una copia del deque lo por detrás hace un ciclo que recorre el deque copiando todos sus valores.

Operación: toString

Complejidad: O(n)

Análisis: La complejidad computacional de la operación se realiza en tiempo O(n) donde n es el tamaño del BigInteger al que se le aplica la operación. La operación recorre el deque<int> deq y va convirtiendo cada numero entero a un string y los concatena a uno por uno.

Sobrecarga: operator ==

Complejidad: Mejor caso O(1) / Peor Caso O(n)

Análisis: La complejidad computacional de la sobrecarga del operador == en su mejor caso se da en tiempo constante ya que en primero lugar compara si el signo de los dos objetos a comparar sean iguales en caso de que la respuesta se daría en tiempo O(1), si los signos son iguales entonces se procede a comparar el tamaño de ambos objetos, si es distinto entonces también se puede resolver en tiempo constante, finalmente si ninguna de las 2 condiciones anteriores se cumple se procede a comparar carácter a carácter, en el mejor caso se puede concluir que los BigInteger no son iguales si detecta que no coinciden en una de sus cifras, y en el peor caso la resolución del algoritmo se da en un tiempo de ejecución O(n) y se da cuando es necesario recorrer por completo ambos objetos.

Sobrecarga: operator <

Complejidad: Mejor caso O(1) / Peor Caso O(n)

Análisis: La complejidad computacional de la sobrecarga del operador < en su mejor caso se da en tiempo constante ya determinar si un objeto es menor que otro se puede realizar con la verificación de unas condiciones de signos y tamaño, en el mejor caso el algoritmo tendría un tiempo de ejecución O(1), en su peor caso tendría que recorrer ambos BigIntegers y verificar carácter a carácter, la complejidad computacional es de O(n) donde n es el tamaño de los BigIntegers si son iguales y hace falta recorrer ambos objetos hasta el final para saber si es menor o no.

Sobrecarga: operator <=

Complejidad: Mejor caso O(1) / Peor Caso O(n)

Análisis: La complejidad computacional de la sobrecarga del operador <= en su mejor caso se da en tiempo constante ya determinar si un objeto es menor que otro se puede realizar con la verificación de unas condiciones de signos y tamaño, en el mejor caso el algoritmo tendría un tiempo de ejecución O(1), en su peor caso tendría que recorrer ambos BigIntegers y verificar carácter a carácter, la complejidad computacional es de O(n) donde n es el tamaño de los BigIntegers si son iguales y hace falta recorrer ambos objetos hasta el final para saber si es menor o igual.

Operación: add

Complejidad: O(n)

Análisis: La complejidad computacional de la operación se realiza en tiempo O(n) donde n es el tamaño de un de los 2 BigInteger dependiendo del caso, la operación en primer lugar compara los signos de los 2 objetos, si estos son iguales entonces se invoca a la operación addLogic la cual tiene una complejidad computacional O(n). De lo contrario, si los signos de los BigIntegers a sumar son distintos entonces, se llama a la operación substractLogic la cual tiene una complejidad computacional O(n).

Operación: substract

Complejidad: O(n)

Análisis: La complejidad computacional de la operación se realiza en tiempo O(n) donde n es el tamaño de uno de los 2 BigIntegers dependiendo del caso. Si los signos de los BigIntegers son iguales entonces se invoca a la operación auxiliar substractLogic la cual tiene un tiempo de ejecución lineal, y de lo contrario se llama a la operación addLogic, que también tiene una complejidad computacional O(n).

Operación: product

Complejidad: O(n^2 x m)

Análisis: La complejidad computacional de la operación O(n x m) donde n es el tamaño del objeto BigInteger al que se le va a aplicar la operación y m es el tamaño del BigInteger que se le pasa a la operación como parámetro. La complejidad de describe como O(n x m) por que por cada cifra del objeto al que se le aplica la operación se multiplican todas las cifras una por una del BigInteger que se pasa en el parámetro. El resultado de esa multiplicación entre BigInteger con una cifra da como resultado un BigIntger que se ira sumando con el resultado de la multiplicación de cada iteración. Al resultado de cada iteración se le aplica la operación auxiliar addCeros la cual adiciona la cantidad de ceros que el número de la iteración en que vaya, por lo que en la última iteración del product se estaría invocando una función que para ese punto tendría una complejidad computacional O(n) solo en la última iteración y a su vez estaría anidad en un ciclo que itera n veces por lo que daría como resultado que la complejidad computacional el O(n^2 x m).

Función: createTable

Complejidad: O(n)

Análisis: Esta función crea un vector que contiene 10 BigIntegers equivalentes a la tabla de multiplicar desde el 0 hasta el 9 del BigInteger que se la pase por parámetro, este algoritmo itera 10 veces en todos sus casos y en cada caso realiza una multiplicación de un BigInteger de tamaño m con otro BigInteger que va a ser un numero entre el 0 y el 9 por lo que el tamaño este es 1, el costo de realizar una multiplicación de n x m^2 donde m es 1, seria igual a tener una complejidad computacional O(n).

Función: findQuotient

Complejidad: O(1)

Análisis: Esta función recibe un objeto BigInteger y un vector que contiene 10 BigIntegers los cuales sería el equivalente a la tabla de multiplicar del 0 al 9 del BigInteger pasado como parametro en la función. La complejidad de esta función es O(1) ya que solo se recorre un vector que siempre va a tener un tamaño constante de 10 elementos y que al encontrar el mayor número que le es de utilidad, se sale del ciclo y no sigue iterando.

Operación: quotient

Complejidad: Mejor caso O(1) / Peor caso O(n^2)

Análisis: La operación quotient tiene como complejidad computacional en su mejor caso una complejidad computacional O(1), esto sucede en el caso en que el objeto BigInteger se divida por un BigInteger que contenga el número 1, como toda división entre el número uno da como resultado su mismo valor, entonces en este caso no se modificaría nada y el tiempo seria constante. Sin embargo en el peor caso la complejidad computacional seria O(n^2) donde n es la resta del tamaño del BigInteger al que se le aplica la operación menos el tamaño del BigInteger que se pasa por parametro,

en cada iteración se crea un BigInteger auxiliar al cual se le van a aplicar product por otro BigInteger de tamaño 1 por lo que la complejidad de esa multiplicación seria O(n x 1^2) y posteriormente se realiza una resta sobre ese BigInteger auxiliar, lo cual toma un tiempo de ejecución O(n). En conclusión, el algoritmo tiene un ciclo que itera n veces y por cada iteración se hace llamado a operaciones con un costo O(n) por lo que la complejidad computacional de la operación Quotient es O(n^2).

Operación: remainder

Complejidad: Mejor caso O(1) / Peor caso O(n^2)

Análisis: La operación remainder tiene como complejidad computacional en su mejor caso una complejidad computacional O(1), esto sucede cuando se quiere aplicar remainder a con un BigInteger que es igual, el residuo en este caso seria 0 por lo que solo hace falta actualizar el deque<int> deq del objeto para que contenga solamente un 0. En el peor caso la complejidad computacional es de O(n^2) ya que se hace necesario realizar el mismo proceso que se da en la operación quotient, a excepción que el valor por el que se modifica el objeto es distinto, en este caso el deque<int> deq se modifica por residuo de la división en vez del cociente.

Operación: pow

Complejidad: O(m x n^3)

Análisis: En la operación pow se eleva un objeto BigInteger a la potencia del número entero que se pasa por parámetro, La complejidad computacional de la operación es O(m x n^3) donde m es el número entero que se le pasa a la operación como parámetro, el algoritmo tiene un ciclo que itera m veces y por cada iteración invoca a la operación producto la cual multiplica a un BigInteger por la copia de el mismo, como la longitud del BigIntger es la misma entonces la complejidad de realizar la operación producto es O(n^3), como la operación product esta anidada en un ciclo que itera m veces, entonces la complejidad del algoritmo seria O(m x n^3).

Sobrecarga: operator+

Complejidad: O(n)

Análisis: En la sobrecarga del operador + se crea una copia del objeto y se le aplica la operación add la cual cuentas con una complejidad O(n)

Sobrecarga: operator-

Complejidad: O(n)

Análisis: En la sobrecarga del operador - se crea una copia del objeto y se le aplica la operación substract la por detrás tiene una complejidad O(n)

Sobrecarga: operator*

Complejidad: O(m x n^2)

Análisis: En la sobrecarga del operador * se crea una copia del objeto y se le aplica la operación product la cual por detrás tiene una complejidad O(m x n^2)

Sobrecarga: operator/

Complejidad: Mejor caso O(1) / Peor caso O(n^2)

Análisis: La sobrecarga del operador / crea una copia del objeto y se invoca a la función quotient la cual por detrás tiene una complejidad O(1) en el mejor caso y O(n^2) el en peor caso.

Sobrecarga: operator%

Complejidad: Mejor caso O(1) / Peor caso O(n^2)

Análisis: La sobrecarga del operador % crea una copia del objeto y se invoca a la función remainder la cual por detrás tiene una complejidad O(1) en el mejor caso y O(n^2) el en peor caso.

Función: sumarListaValores

Complejidad: O(m x n)

Análisis: La complejidad computacional de la función es O(m x n) donde m es la cantidad de elementos de la lista de BigIntegers y n es el tamaño de un BigInteger. La complejidad se puede predecir como O(m x n) porque la función tiene un ciclo que itera m veces y en cada iteración se invoca a la operación add la cual tiene una complejidad de O(n) donde n es el tamaño del BigInteger.

Función: multiplicarListaValores

Complejidad: O(m x k x n^2)

Análisis: La complejidad computacional de la función es O(m x k x n^2) donde m es la cantidad de elementos de la lista de BigIntegers, en cada iteración se invoca a la operación product la cual tiene un complejidad computacional O(k x n^2) donde k es el tamaño del objeto BigInteger al que se le aplica la operación producto y n es el BigInteger pasado por parámetro a la operación producto.